Dear Customer,

First, let me thank you for purchasing our Catch Game Template. We are truly grateful and humbled by your purchase. We hope this package aids you in your development process as it has ours.

From Everyone at You Contribute Games

Purpose:

The Catch Game Template is designed for the purpose of shortening the development timeline when developing an egg catch styled game or mini game in Unity. It was built in such a manner that it can be used to create 2D and 3D games. With its modular design many of its components can be used in other facets of your game design process as well. This document is intended to help you in using this template to its fullest by providing insight into how the project works and how it can be tweaked and adjusted to your liking.

If you have questions:

If you have any questions please feel free to email us at CustomerService@YouContributeGames.com

Find us on the web:

- www.YouContributeGames.com
- www.facebook.com/YouContributeGames
- www.youtube.com/YouContributeGames
- www.twitter.com/YCGNews @YCGNews
- www.twitch.tv/YCGonTwitch

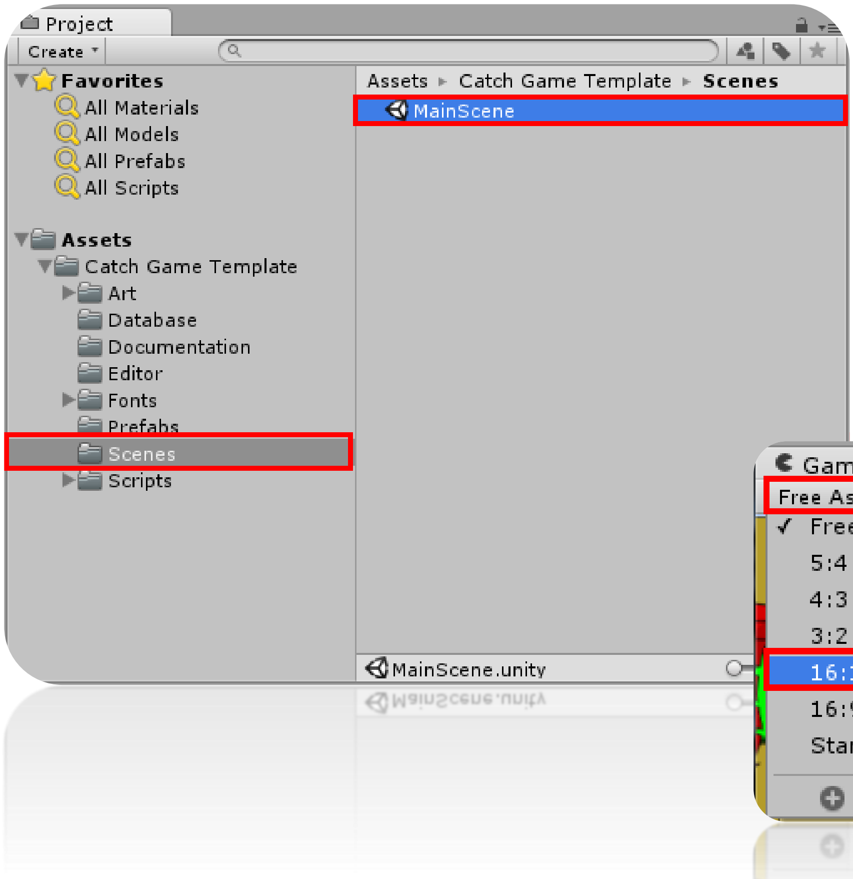
Game Concept:

- Catch all of the falling objects, but be careful to only catch the good ones
- If you catch a bad object your health bar will drop
    - If it drops to zero the game is over
- As you catch good objects your score will increase
    - There are different types of good objects and each has a different score value
- When lots of items are falling try to go for the ones that give you the highest score
- See how high you can get your score.

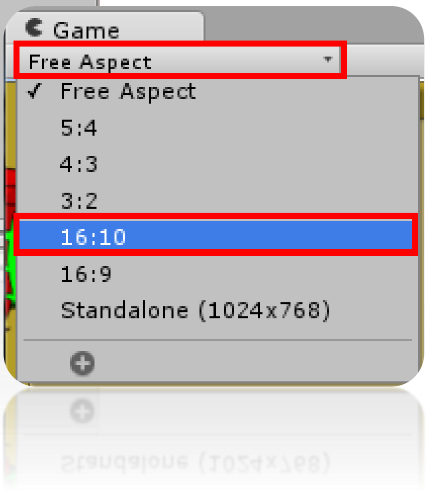
Setting up the project

- Import the Catch Game Template asset package
- Navigate to the Scenes folder and open the MainScene scene (Screen Shot 1)
- In the Game window select 16:10 in the resolution drop down (Screen Shot 2)
- Press play button to test the package
- Update art and scripting as needed to create your awesome game!

Screen Shot 1



Screen Shot 2



Scripts

All scripts in the Catch Game Template utilize a standardized layout. We have included Template.cs if you wish to use the template in your own scripts. The purpose of this script is to provide an easy way to know the general area that certain things will be located in, by grouping them into 5 regions. All 5 regions are present in each script and if that region is empty you know that type of data is not present in the script. For a detailed explanation of our template and how it is laid out please take a look at our blog post detailing it in full.

Template Blog Post: http://www.youcontributegames.com/coding-best-practice/scripttemplate/

All scripts are commented in detail to give definitions on how each variable is used and in a way reads like pseudo code as you follow through the script. Below is a high level explanation of each script. There is no better way to know how code works than looking at it directly. We have worked hard to over comment the code so that it can easily be followed. Where this document gives you the broad picture of each script the comments will give you the deep dive.

Manager Scripts

Manager Scripts are designed to manage specific functionalities of the game. They are laid out so that they could easily be modified to fit into other games. As example, HealthManager.cs could be imported into any game to control a player's health and their health bar slider.

GameManager.cs

GameManager.cs is the primary game state controller. It handles switching between the game states which include Main Menu, Play, and High Score Menu. It also manages updating the top ten high scores in the high score menu each time a game ends.

The core functionality of changing game states is handled through the use of a case statement where each state has a numeric value and all components are activated or deactivated in the scene hierarchy dependent on which state is being called.

CollisionManager.cs

CollisionManager.cs manages all collision detection in the template. There are only 3 things that could possibly collide with anything in the standard template setup. Those are the Falling Objects, the bucket, and the ground. The falling objects can collide with the bucket and ground but the bucket and ground cannot collide with each other in their standard setup. Because of this it is understood that the bucket and ground need to determine if a falling object collided with them.

The collision logic is therefore setup so the same script can be used on both the bucket and the ground. The collision logic checks:

- To see if it is on the Bucket or Ground using the tag assigned to the object
- Then if it was collided into by a good egg or bad egg
    - Good Eggs are good falling objects that give you points
    - Bad Eggs are bad falling objects that reduce your health.

Once a collision has been detected and the logic in this script determines what collided with what, it relays messages to the appropriate place indicating what should happen because of the collision, like add to the score or subtract from the health. It also tells the falling objects that they have been caught or hit the ground so they can be reset and available for the spawning logic to use again later.

By updating the IF statements in the OnTriggerEnter() method you can adjust how the game behaves when objects collide. You will notice that the SendMessage() statements are commented out when a falling object hits the ground. You can:

- Uncomment these for a whole new play style because now
    - if you don't catch Good Eggs your score will drop when they hit the ground
    - If you avoid catching Bad Eggs when they hit the ground your score will increase
- Encapsulate this new logic in an IF statement that checks to see if you are at a certain level if you want to only have this happen once the player is further into the game.

These are just some ideas to get your creative juices flowing around how you can use this template to create your own vision.

LevelManager.cs

LevelManager.cs controls all things level related. This script is very modular and could easily be used for managing the current level of any arcade style game.

- It calculates what score is needed to move to the next level
- It tracks the score in ScoreManager.cs to see if the current score meets the threshold to increase the level
- It has an override method that allows you to manually set the current level
    - This is handy since the rate at which objects fall is determined by the level and you can tweak how fast objects fall in the background while in menus

HealthManager.cs

HealthManager.cs controls the:

- Players Health Value
- Controls the Health Bar in the Heads Up Display(HUD)
- Tracks the games lose condition which is the players health dropping to zero(0)
- Controls the reduction of the players heath when notified by other objects to do so
    - Example – the Collision Manager detects that a Bad Egg was caught in the Bucket and sends notice to reduce the players health

SpawnManager.cs

SpawnManager.cs controls which objects fall and from where. When initialized the Spawn Manager builds a pool of objects it can spawn. By pooling these objects rather than creating and deleting new ones each time this reduces the needed resources for garbage cleanup. This prevents the resource usage from spiking at unwanted times which could lead to laggy gameplay.

PlayerManager.cs

PlayerManager.cs controls how the player moves. It utilizes both Update() and FixedUpdate() to actually push the player around and not just change their position on screen so that you can get true physics reactions when it collides with things. Since Update() does not happen on a fixed cycle, FixedUpdate() is used to get the input location and call for the player to move. Things such as is the player touching the screen don't need to happen each physics loop and can therefore be checked in the Update() cycle.

ScoreManager.cs

ScoreManager.cs handles updating the score in the heads up display for the player to see and updating the high score list in the data manager which feeds the high score menu. Although ScoreManager.cs holds the score variable during gameplay, this variable is setup as a public static variable to easily be updated by other objects. Another way to set this up would be to create an update score method and have the falling objects call this sending it the value to change by. You can decide which method will work best for your game.

Falling Object Scripts

We have opted to give each type of falling object their own script. The falling objects script will each track its own position to see if for some reason it has left the play area or gotten stuck in the spawn area and reset itself as a safety measure.

Bolts.cs

Bolts.cs is applied to each bolt in the game. It will randomly set itself to one of the available bolt color options and assign itself the appropriate score value to match. Based on the color option it will also update its sprite image. If it is informed by the Collision Manager it can update the Score Managers score value appropriately.

Nuts.cs

Nuts.cs is applied to each nut in the game. It will randomly set itself to one of the available nut color options and assign itself the appropriate score value to match. Based on the color option it will also update its sprite image. If it is informed by the Collision Manager it can update the Score Managers score value appropriately.

Screws.cs

Screws.cs is applied to each screw in the game. It acts in the same manner as Bolts.cs and Nuts.cs except for the fact that it does not adjust the visual or score value of itself like they do.

Background Scripts

ConstantParallax.cs

ConstantParallax.cs manages the pretty floating clouds. In the demo scene setup there are 3 planes with a single cloud image applied as a material to it. Constant Parallax simply adjust the position of the material on those planes shifting their position as time progresses giving the appearance that they are moving across the screen. This works well for games such as this one where the play area never truly moves.

The Data Manager

The Data Manager is how we create the games internal database that stores the player's high scores between play sessions. There are four primary components to the Data Manager. Those are DataClass.cs (the blueprint), DataManager.cs (the map to the building), DataManagerEditor.cs (the builder), and Data.asset (the finished storage building).

DataClass.cs

DataClass.cs is the blue print for our storage building. It is a class that extends from ScriptableObject rather than MonoBehavior like most scripts. By extending ScriptableObject, when instantiated (created) it exists without being tied to a GameObject in the scene hierarchy. This means it exists outside of any single scene or play session so any data stored in it is for all intents and purposes saved (until they are deleted by you).

DataManager.cs

DataManager.cs is the container in our scene that holds the reference to our data object so we can reference it in other scripts.

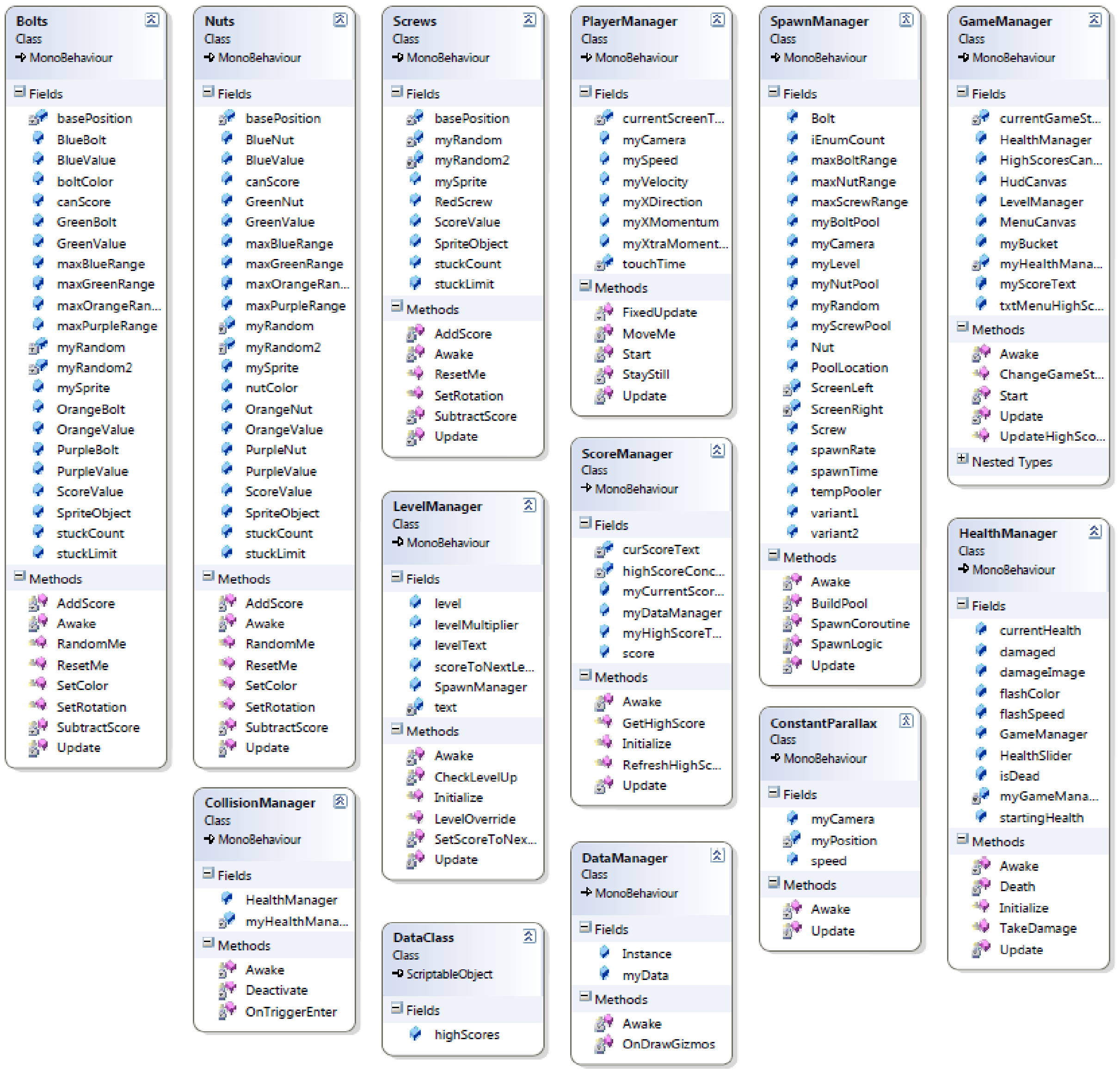
DataManagerEditor.cs

DataManagerEditor.cs is what actually builds the storage asset for us. It extends from editor and is the custom editor for the DataManager.cs and therefore must reside in your games editor folder. It holds the path that our Data Object should be saved in and file name it should be called when created. Anytime the object holding the DataManager.cs script is enabled (clicked on) in the scene it will check to see if that instance of DataManager.cs already has a Data Object referenced in its myData variable. If it does not it will create a new one.

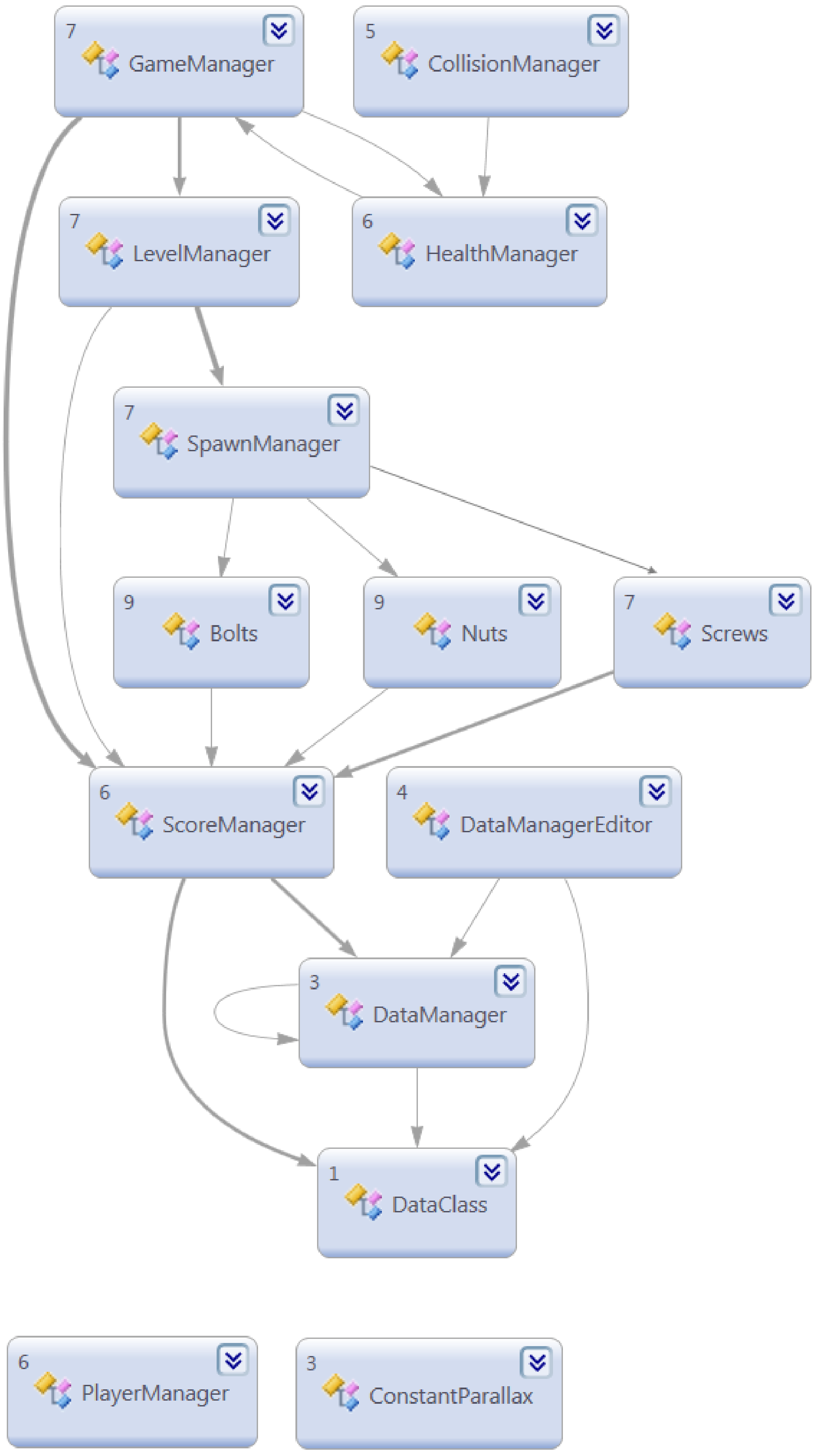
Data.asset

Data.asset is not actually a class but is just an instantiated version of DataClass.cs containing all the variables defined in that class. If you were to add additional variables to DataClass.cs you could create a new Data.asset with those variables available in your game. You could also change the name from Data.asset in the dataPath variable of DataManagerEditor.cs to create a new asset for storing data. Maybe you could track the highest level the player has reached or anything else your creativity calls for.

Class Diagrams



Class Dependencies


7 GameManager
5 CollisionManager
7 LevelManager
6 HealthManager
7 SpawnManager
9 Bolts
9 Nuts
7 Screws
6 ScoreManager
4 DataManagerEditor
3 DataManager
1 DataClass
6 PlayerManager
3 ConstantParallax